Süsteemprogrammeerimine keeles C

Loeng 10

Milles räägitakse sellest, mis on signaalid ning tehakse põige kompileerimise abivahendite "make", "automake" ja "autoconf" põnevasse maailma.

Eelmises osas (Varieeruvad argumendid)

Varieeruva argumentide hulgaga funktsioonis kasutatakse kolme punkti
foo(esimene, teine, ...) { /* sisu */ }

va_list arglist;
va_start(arglist, teine);
va_arg(arglist, *tüüp*);
va_end(arglist);

man vararg või man stdarg

Eelmises osas (Protsessid)

Protsess käivitatavas failis: HEADER, TEXT (Program), DATA (Initialized), BSS (= uninitialized data), RELOCATION, SYMBOL TABLE

Protsess mälus: TEXT, DATA, (BSS), free mem, STACK, USER BLOCK (kernelis)

Eelmises osas (exec())

exec() perekonna käsud asendavad parajasti jooksva protsessi failist käivitatud protsessiga

execl(), execv(), execle(), execve(), execlp(), execvp() vastavalt argumentide kujule

argumentideks käivitatav fail, argumendid, keskkond

argumendid: 1. argument peab olema programminimi
keskkond: stringid kujul TÄHIS=VÄÄRTUS

Eelmises osas(fork)

fork() dubleerib parajasti jooksva protsessi

Vanemprotsess saab lapsprotsessi PID

Üks väljakutse, kaks *return*i

Eelmises osas (wait)

Ootab lapsprotsessi lõpetamist või tagastab viimase lõpetanud lapsprotsessi andmed

Lõpetab zombiprotsessid

Eelmises osas (exit)

exit() lõpetab viisakalt programmi töö: sulgeb deskriptorid, flushib streamid

atexit() võimaldab määrata normaalse väljumise puhul jooksutatavaid funktsioone

Eelmises osas (Deemonid)

Deemon on programm, mis töötab taustal.

Olemuselt disainimuster

Taustale saab programmi saata nii, et vanemprotsess lõpetab töö, fork abil tehtud lapsprotsess teeb kasulikku tegevust.

Eelmises osas (omanik & PID)

Protsessil on tegelik omanik ja *kehtiv* (*effective*) omanik.

Esimene neist ütleb kes tegelikult jooksutab, teine neist seda, kelle õigustes programm faile avab.

Samamoodi on grupiga

PID on protsessi number. Init on alati protsess nr 1

Tänases osas

- Protsesside juhtimine
- Long jump
- Make
- Automake & autoconf

Protsesside juhtimine

- Töötavaid protsesse saab signaalidega mõjutada.
- Signaal on tarkvaraline katkestus, mida on võimalik ettemääratud tingimustel protsessidele saata.
- Signaaliga saab protsess teha 3 asja:
  - Ignoreerida (SIGKILL ei ole ignooritav)
  - Jätta signaal haldamata, mis enamasti tähendab protsessi lõpetamist. Mõned signaalid annavad ka *core* välja.
  - Signaal kinni püüda: anda juhtimisjärg üle ettemääratud programmijupile. SIGKILL ja SIGSTOP pole püütavad.

Signaal

- Signaali tüüpi kirjeldab täisarv.
- Kuna täisarvu meeldejätmine on väheotstarbekas, vastab igale signaalile mingi konstant (algavad nad prefiksiga SIG).
- Näiteks:
  - SIGKILL, SIGQUIT, SIGTRAP ja meie kõigi lemmik SIGSEGV - Segmentation fault

Signaalide väärtused (Solaris)

SIGHUP	1	Exit	Hangup (see termio(7I))
SIGINT	2	Exit	Interrupt (see termio(7I))
SIGQUIT	3	Core	Quit (see termio(7I))
SIGILL	4	Core	Illegal Instruction
SIGTRAP	5	Core	Trace or Breakpoint Trap
SIGABRT	6	Core	Abort
SIGEMT	7	Core	Emulation Trap
SIGFPE	8	Core	Arithmetic Exception
SIGKILL	9	Exit	Killed
SIGBUS	10	Core	Bus Error
SIGSEGV	11	Core	Segmentation Fault
SIGSYS	12	Core	Bad System Call
SIGPIPE	13	Exit	Broken Pipe
SIGALRM	14	Exit	Alarm Clock
SIGTERM	15	Exit	Terminated
SIGUSR1	16	Exit	User Signal 1
SIGUSR2	17	Exit	User Signal 2
SIGCHLD	18	Ignore	Child Status Changed

+ veel

signal()

- Signaalidega ümberkäimiseks on järgmine käsk:

```
#include <signal.h>
void *signal(int signum, void (*handler)(int));
```

- signal() käsk määrab signaali numbriga signum saabumisel käivitatava funktsiooni, mis võib olla:
  - kasutaja poolt kirjutatud funktsioon
  - SIG_IGN : ignoreeri signaali
  - SIG_DEFL : taasta vaikimisi käitumine
- Täisarvuline argument *handler*ile on signaali number: ühe funktsiooniga saab hallata mitut erinevat signaali.

signal() (2)

- Signal taastab handleri eelneva väärtuse või SIG_ERR vea korral

```
extern int handler(int);
signal(SIGALRM, handler);
signal(SIGINT, SIG_IGN);
```

- Unix saadab protsessile signaali, kui midagi juhtub: näiteks kui aritmeetikas tekib ületäitumine, terminalil vajutatakse katkestusnupule või kui käivitatakse vigaseid käske. Protsessile saab signaale saata süsteemikäsuga kill()

kill()

```
int kill(pid_t pid, int sig);
```

- Võimaldab saata signaali igale protsessile või nende grupile
- Positiivse pid korral, saadetakse signaal vastavale protsessile
- Kui pid on 0, läheb signaal kõigile antud grupi protsessidele: näiteks ühelt terminalilt käivitunud protsessidele
- Kui pid on -1, saadetakse signaal kõigile protsessidele peale kõige esimese

kill() (2)

- Kui sa pole juurkasutaja, läheb -1 korral signaal ainult kill() käsku välja kutsuva protsessiga sama uid-ga protsessidele
- Kui pid on väiksem kui -1, läheb signaal kõigile protsessidele grupis -pid (**NB!** miinusmärk!)
- Kui sig on 0, ei saadeta signaali välja, aga veakontrollid samas töötavad
- Edu korral tagastub 0, vea korral -1 ja määratakse errno

alarm()

- Määrab kellaajalise signaali

```
unsigned int alarm(unsigned int seconds);
```

- seconds sekundi pärast antakse protsessile SIGALRM signaal
- Kui sekundid on 0, ei määrata uut alarmi
- Igal juhul tühistatakse plaanis olev alarm
- Tagastab mitme sekundi pärast eelmine alarm oleks toimunud, või 0, kui ühtki plaanis polnud

Long Jump

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

- setjmp ja longjmp aitavad katkestuste haldamise ajal programmi teises kohas jätkama panna.
- setjmp() salvestab *stack*i sisu/keskkonna env puhvrisse, et seda hiljem longjmp() väljakutsel kasutada. Kui setjmp() välja kutsunud funktsioon tagastub, kaotab env sisu oma mõtte.
- setjmp() tagastab 0, kui tagastub esimest korda ja nullist erineva väärtuse, kui tagastub pärast longjmp() väljakutset

Long Jump (2)

```
void longjmp(jmp_buf env, int val);
```

- longjmp() taastab keskkonna, milles setjmp() välja kutsuti. Pärast longjmp() väljakutset käitub programm justkui setjmp() oleks tagastanud väärtuse val. longjmp() abil ei ole võimalik 0 väärtust saata: see asendub väärtusega 1

```
jmp_buf env;
if ((val=setjmp(env)) == 0)
   printf("Now we have set long jump\n");
else
   printf("Long jump has returned value
%d\n",val);
..........…
longjump(env, 3);
```

long.c

Make

- Make lahendab mitu probleemi:
  - Suurte mitmest osast koosnevate programmide kompileerimise juhtimine
  - Pakub võimaluse programmi kompileerimiseks ilma ehitusprotsessi süüvimata
  - Programmeerija mugavus: programmi ehitamine peab olema atomaarne tegevus

Mitu lähtekoodifaili

- Suured projektid ei taha hästi ühte faili ära mahtuda.
- C pakub mitut tehnikat, mis suured projektid hallatavamaks teevad
    - keegi kuskil ei valva, et neid reegleid täidetaks, aga kõik head C programmeerijad oskavad seda teha
- Programm jaotub mooduliteks
  – Mooduli *header* (.h) fail sisaldab prototüüpe
  – .c fail sisaldab funktsioonide definitsioone
- Moodulid kompileeritakse sõltumatult ja lingitakse

C ja H

- C failid
  - sisaldavad lähteteksti ja globaalmuutujate definitsioone
  - kompileeritakse korra, include käskudes ei kasutata
- H failid
  - kirjeldavad ära mooduli liidese, "kirjeldavad" .c faile
  - tüüpide ja structide deklaratsioonid
  - const ja #define
  - #include käsud teiste *header*ite kaasamiseks
  - funktsiooniprototüübid

Make

- Suuri mitmest moodulist koosnevaid programme on käsitsi kompileerida väga paha
- Seda probleemi lahendab Make

```
# Makefile for the sample
sample:  sample.o  my_math.o
        cc –o sample sample.o my_math.o
sample.o:  sample.c  my_math.h
        cc –c sample.c
my_math.o:  my_math.c my_math.h
        cc –c my_math.c
clean:
        rm sample *.o core
```

NB! Taandeks on [tab], mitte tühikud!

Makefile

- Programmi kataloogi salvestatakse Makefile (või makefile)
- Iga kord kui soovid programmi kompileerida, anna käsk: "make"
- Make:
    - leiab makefile'i
    - kontrollib reeglid ja sõltuvused: mis vajab täiendamist
    - täiendab need komponendid, mida vaja on
- Näide:
  - Kui sample.c on uuem kui sample, antakse vaid käsud:
    - cc -c sample.c
    - cc -o sample sample.o my_math.o

Makefile (2)

- Loodud failide puhastamiseks anda käsk:

```
make clean
```

- Uuesti kompileerimiseks:
  - Eemaldada kõik genereeritud failid, et neid uuesti teha

```
make clean; make
```

  - Või võid anda käsu:

```
touch my_math.h
```

    - seejärel jooksuta make'i
    - touch täiendab faili my_math.h muutmise aega ja sellest järeldab make, et tarvis on täiendamist

Reeglid

- Lihtsas *makefile*is ongi ainult reeglid. Nad on kujul:

```
target: prerequisites ...
        command
        ...
```

  - target e. eesmärk. Enamasti loodava faili nimi, kuid võib olla ka tegevus: näiteks "clean"
  - prerequisite e. eeldus on fail või käsk, millest eesmärk sõltub: see, mis on tema loomiseks tarvilik
  - command on tegevus, mis make teeb, et eeldusest eesmärk valmis teha. Ära tab-i unusta!
  - Esimene reegel failis on vaikimisi täidetav: käivitub kui make teisi argumente ei saa

Eeldusreeglid

- Make saab objektfailiga hakkama ka reeglitega, kus on puudu käsud. Lihtsal puhul pole igast .c failist .o faili tegemiseks käsku eraldi tarvis välja kirjutada.
- Kui .c faili sedasi kasutada, lisatakse see eeldustesse automaatselt. Kui me .o tegemisel käsku ei kirjelda, ei maini me ka .c faili

Eeldusreeglite näide

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

# Note special .PHONY keyword here!!!
.PHONY : clean
clean :
        rm edit $(objects)
```

.PHONY

- Make clean töötab väga hästi, sest tal pole eeldusi ja fail clean puudub
- Kui kataloogi peaks tekkima fail nimega clean, vaatab make, et kõik on olemas ja käske ei täideta.
- .PHONY reegel ütleb, et eesmärk ei ole fail, vaid mingi tegevus

```
.PHONY: clean
clean:
        rm -f *.o
```

Makrod ja *makefile*id

- Makrod on võimalus asendada pikki korduvaid tekstijuppe mugavamate lühikeste muutujanimedega ning kogu protsess vähem arusaadavamaks, ent paremini hallatavaks muuta
- Makrod defineeritakse kujul nimi = midagi
- Makrosid kasutatakse kujul $(nimi) või ${nimi}

Make ja mitu kataloogi

- Kui program on suurem, on ta mõistlik moodulite kaupa kataloogidesse jagada. Keerukus kasvab.
- Iga mooduli jaoks tehakse oma makefile
- Programmil on iga mooduli jaoks kataloog ja üks kataloog *kõigi* .h failide hoidmiseks
- Programmi makefile tegeleb programmi loomisega
- Moodulites olevad makefile'id teevad valmis moodulite objektfailid

Mitme kataloogi näide

- C programm koosneb moodulitest Stack ADT, Queue ADT ja main.
- Programmil on 7 faili: StackTypes.h, StackInterface.h, QueueTypes.h, QueueInterface.h, StackImplementation.c, QueueImplementation.c ja Main.c
- Eesmärgiks on teha programm kataloogis Assn, millel on alamkataloogid Stack, Queue, Main ja Include
- Kõik 4 .h faili on kataloogis Include

Mitme kataloogi näide (2)

- Stack sisaldab faili StackImplementation.c ja järgmist *makefile*i:

```
export: StackImplementation.o

StackImplementation.o: StackImplementation.c \
                       ../Include/StackTypes.h \
                       ../Include/StackInterface.h
          gcc -I../Include -c StackImplementation.c
   # substitute a print command of your choice for lpr below
   print:
          lpr StackImplementation.c
   clean:
          rm -f *.o
```

Mitme kataloogi näide (3)

- Queue sisaldab QueueImplementation.c ja järgmist *makefile*i:

```
export: QueueImplementation.o
   QueueImplementation.o: QueueImplementation.c \
                          ../Include/QueueTypes.h \
                          ../Include/QueueInterface.h
            gcc -I../Include -c QueueImplementation.c
   # substitute a print command of your choice for lpr
     below
   print:
            lpr QueueImplementation.c
   clean:
            rm -f *.o
```

Mitme kataloogi näide (4)

- Märkus: -I (suur i) võti ütleb kompilaatorile #include lausetega antud .h failide otsimiseks kohti.
- Mitme kataloogi määramiseks võib vahele panna komasid. Pane tähele, et tühikut ei kasutatud
- -I võtmega saame kõik .h failid ühte kohta panna ja ei pea neid igasse kataloogi eraldi panema

- Sümbol \ enne eesmärk/eeldus realõppu tühistab realõpu tähenduse reeglite alustamise märgina ja võimaldab eeldusi mitmel real

Mitme kataloogi näide (5)

- Kataloog Main sisaldab faili main.c ja *makefile*i:

```
 export: Main
 Main: Main.o StackDir QueueDir
        gcc -o Main Main.o ../Stack/StackImplementation.o \
                           ../Queue/QueueImplementation.o
Main.o: Main.c ../Include/*.h
        gcc -I../Include -c Main.c
StackDir:
        (cd ../Stack; make export)
QueueDir:
         (cd ../Queue; make export)

#jätkub järgmisel lehel...
```

Mitme kataloogi näide (6)

```
print:
        lpr Main.c
printall:
        lpr Main.c
        (cd ../Stack; make print)
        (cd ../Queue; make print)

clean:
        rm -f *.o  Main  core
cleanall:
        rm -f *.o  Main  core
        (cd ../Stack; make clean)
        (cd ../Queue; make clean)
```

Mitme kataloogi näide (7)

- Kui Unixi käskude jada panna sulgudesse ( ) , käivitatakse tegevus alamprotsessina ja käsud käivituvad selle protsessi raames.
- Näiteks kui käsk (cd ../Stack; make export) käivitub, läheb alamprotsess kataloogi Stack, käivitab make käsu; kui protsess lõpetab, jätkab vanemprotsess vanas kataloogis; lisa-cd käsku pole tarvis

Makefile uuesti

- Võime Stack kataloogi faili ümber kirjutada nii:

```
CC = gcc
HDIR = ../Include
INCPATH = -I$(HDIR)
DEF =  $(HDIR)/StackTypes.h  $
  (HDIR)/StackInterface.h
SOURCE = StackImplementation
export:  $(SOURCE).o

$(SOURCE).o:  $(SOURCE).c  $(DEF)
            $(CC)  $(INCPATH)  -c  $(SOURCE).c
print:
            lpr  $(SOURCE).c
clean:
```

Muud võimalused

- Uuemad make versioonid nagu GNU ja Solarise omad teevad ka järgmisi trikke:
  - kontrollstruktuurid ja tingimuslikud laused & tsüklid
  - lihtsad tekstimuundamisfunktsioonid
  - automaatsed Makefile elementidele viitavad muutujad nagu eesmärgid ja sõltuvused
- gmake jaoks on manuaal siin: http://www.gnu.org/software/make/manual/make.html

GNU autotools

- http://sources.redhat.com/autobook/autobook/autobook_toc.html
- Makefile ei ole hea kui on vaja teha porditavat programmi erinevatele Unixitele
- automake, autoconf
- Programmeerija kirjutab Makefile.am ja configure.in failid
- automake teeb Makefile.am failist Makefile.in faili
- autoconf teeb configure.in failist configure (shelliskript)
- configure teeb Makefile.in abil Makefile

Makefile.am

- Näidis:

```
## Makefile.am -- Process this file with automake to
produce Makefile.in
bin_PROGRAMS = foonly
foonly_SOURCES = foo.c foo.h nly.c scanner.l parser.y
foonly_LDADD = @LEXLIB@
```

configure.in

- Näidis:

```
dnl Process this file with autoconf to produce a
configure script.

AC_PREREQ(2.59)

AC_INIT([foonly], [2.0], [gary@gnu.org])

AM_INIT_AUTOMAKE([1.9 foreign])

AC_PROG_CC
AM_PROG_LEX
AC_PROG_YACC

AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Autotoolsi kasutamine

- Tavailne töö käik:

```
aclocal
autoconf
automake
./configure
make
make install
```

- Jagamiseks

```
make dist
```

  - valmistab valmis konfiga xxx.tar.gz faili

Teretulemast loengu lõppu

Slaide selles loengus: 47